deep se
dependable evolvable pervasive software engineering group

Classes

Danilo Ardagna

Politecnico di Milano
danilo.ardagna@polimi.it

POLITECNICO DI MILANO

Content

- friends
- static members
- Class scope

friends

- Members defined after a **public** specifier are accessible to all parts of the program
  - public members define the interface to the class
- Members defined after a **private** specifier are accessible to the member functions of the class but are not accessible to code that uses the class
  - private sections encapsulate (i.e., hide) the implementation
- A class can allow another class or function to access its nonpublic members by making that class or function a **friend**

friends

```
class Sales_data {// All code in Sales_data.h
// friend declarations for nonmember Sales_data operations added
friend Sales_data operator+(const Sales_data&, const Sales_data&);
// other members and access specifiers as before
public:
        std::string isbn() const { return bookNo; }
        Sales_data& operator+=(const Sales_data&);
private:
        std::string bookNo;
        unsigned units_sold = 0;
        double revenue = 0.0;
};
// declarations for nonmember parts of the Sales_data interface
Sales_data operator+(const Sales_data&, const Sales_data&);
```

in-class we're declearing the friend, out-of-class we're declearing again the function (not as friend anymore)

In-class it **not** declaring the function, it's declaring the friendship. That's why the "true" declaration is performed outside.

friends

- A friend declaration only specifies access. It is not a general declaration of the function
  - If we want users of the class to be able to call a friend function, then we must also declare the function separately from the friend declaration
  - We usually declare each friend (outside the class) in the same header as the class itself
  - This is why our Sales_data header provides a separate declaration (aside from the friend declaration inside the class body) for operator+

operator+ implementation (declared as friend) in Sales_data.cpp

```
Sales_data operator+(const Sales_data& lhs,
                     const Sales_data& rhs)
{
    Sales_data ret;

    ret.bookNo =  lhs.bookNo;
    ret.units_sold = lhs.units_sold + rhs.units_sold;
    ret.revenue = lhs.revenue + rhs.revenue;

    return ret;
}
```

we can access directly also the private members!

operator+ implementation (as plain helper function)

```
class Sales_data {// All code in Sales_data.h

// other members and access specifiers as before
public:
        std::string isbn() const { return bookNo; }
        Sales_data& operator+=(const Sales_data&);
private:
        std::string bookNo;
        unsigned units_sold = 0;
        double revenue = 0.0;
};
// declarations for nonmember parts of the Sales_data interface
Sales_data operator+(const Sales_data&, const Sales_data&);
```

Same as before but without the friend inside the class → HELPER (≠ FRIEND)

operator+ implementation (as plain helper function) in Sales_data.cpp

```
Sales_data operator+(const Sales_data& lhs,
                     const Sales_data& rhs)
{
    Sales_data ret;

    ret.set_bookNo(lhs.isbn());
    ret.set_units_sold(lhs.get_units_sold() +
                       rhs.get_units_sold());
    ret.set_revenue(lhs.get_revenue()  +
                    rhs.get_revenue());

    return ret;
}
```

A HELPER is not a FRIEND and so it cannot access the private members directly, it has to use the setter/getter.

static Class Members

static Class Members

- Classes sometimes need members that are associated with the class, rather than with individual objects of the class type
- For example, a bank account class might need a data member to represent the current prime interest rate
- In this case, we'd want to **associate the rate with the class, not with each individual object**
  - From a memory efficiency standpoint, there'd be no reason for each object to store the rate
  - Much more importantly, if the rate changes, we'd want each object to use the new value

static Class Members

- We say a member is associated with the class by adding the keyword `static` to its declaration
- Like any other member, static members can be `public` or `private`
- The type of a `static` data member can be `const`, reference, array, class type, and so forth

static Class Members

```
class Account {
public:
      void calculate() { amount += amount * interest_rate; }
      static double rate() { return interest_rate; }
      static void rate(double);
private:
      std::string owner;
      double amount;
      static double interest_rate;
      static double init_rate();
};
```

getter and setter for a static member (function overload (we can do it because the params are ≠))

Static member functions:
- Are not bound to any object
- Do not have a `this` pointer

for example, this interest_rate is associated with the class, not with the object

A declaration like this:

static double rate() const;

doesn't make any sense!!!

we say static function to all the functions that rely only on the static members

"const" is for objects ("const" protects objects from changing, but a static method/member is associated with CLASSES NOT OBJECTS)

"this" is a pointer to the underlying object on which we're running the function STATIC FUNCTIONS ARE NOT ASSOCIATED WITH OBJECTS

static Class Members

BEST SYNTAX

- We can access a static member directly through the scope operator:

```
double r;
r = Account::rate(); // access a static member using the
                     // scope operator
```

- Even though static members are not part of the objects of its class, we can use an object, reference, or pointer of the class type to access a static member:

```
Account ac1;
Account *ac2 = &ac1;
// equivalent ways to call the static member rate function
r = ac1.rate(); // through an Account object or reference
r = ac2->rate(); // through a pointer to an Account object
```


static Class Members

- Member functions can use static members directly, without the scope operator:

```
class Account {
public:
      void calculate() { amount += amount * interest_rate;
}
      // remaining methods as before
private:
      static double interest_rate;
      // remaining members as before
};
```

not static, associated with the object!

static

static Class Members

- As with any other member function, we can define a static member function inside or outside of the class body
- When we define a static member outside the class, we do not repeat the static keyword. The **keyword appears only with the declaration inside the class body**:

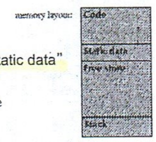
```
void Account::rate(double new_rate)
{
      interest_rate = new_rate;
}
```

`static` Class Members

- Because static data members are not part of individual objects of the class type, they are not defined when we create objects of the class. As a result:
  - they are not initialized by the class constructors
  - we may not initialize a static member inside the class
  - we must define and initialize each static data member outside the class body
  - like any other object, a static data member may be defined only once
- Like global objects, static data members are defined outside any function
  - once they are defined, they continue to exist until the program completes

The computer's memory

- As a program sees it
  - Local variables "live on the stack"
  - Global variables **and static members** are "static data"
  - The executable code is in "the code section"
  - "Free store" is managed by `new` and `delete`



`static` Class Members

- We define a static data member similarly to how we define class member functions outside the class:
  - name the object's type, followed by the name of the class, the scope operator, and the member's own name:

```
// define and initialize a static class member
double Account::interest_rate = init_rate();
```

- The best way to ensure that the static members are defined exactly once is to put the definition of static data members in the **same file that contains the definitions of the class non-inline member functions**

`static` Class Members

- We define a static data member similarly to how we define class member functions outside the class:
  - name the object's type, followed by the name of the class, the scope operator, and the member's own name:

```
// define and initialize a static class member
double Account::interest_rate = init_rate();
```

- The best way to ensure that the static members are defined exactly once is to put the definition of static data members in the **same file that contains the definitions of the class non-inline member functions (.cpp file!)**

Class scope

Scope

- A scope is a region of program text
  - Global scope (outside any language construct, e.g., before `main()`)
  - Local scope (between { … } braces)
  - Statement scope (e.g. in a for-statement)
  - **Class scope (within a class)**
- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name ("can't look ahead" rule)
  - **Exception to this rule: class members can be used within the class before they are declared**
- A scope keeps "things" local
  - Prevents my variables, functions, etc., from interfering with yours
  - Remember: real programs have many thousands of entities
  - Locality is good!
    - Keep names as local as possible

Another example - Scopes nest

```
int x;  // global variable – avoid those where you can
int y;  // another global variable

int f();

int main(){
        x =8; y = 3;
        f();
        cout << x << ' ' << y << '\n';
}

int f()
{
  int x;                // local variable (Note – now there are two x's)
  x = 6;                // local x, not the global x
  {
        int x = y;      // another local x, initialized by the global y
                        // (Now there are three x's)
        ++x;
  }
  // what is the vale of x here?
  y++;
}
```

DEMO

// avoid such complicated nesting and hiding: keep it simple!

Scope

```
// get max and abs from algorithm and cstlib
// no r, i, or v here
class My_vector {
public:
  int largest()                              // largest is in class scope
  {
        int r = 0;                           // r is local
        for (int i = 0; i<v.size(); ++i)     // i is in statement scope
             r = max(r,abs(v[i]));
        // no i here
        return r;
  }
  // no r here
private:
  vector<int> v;                             // v is in class scope

};
// no v here
```

(Class exceptions:)
here we're using/accessing v BEFORE we declare v

Scope

```
// get max and abs from algorithm and cstlib
// no r, i, or v here
class My_vector {
public:
  int largest_buggy()                        // largest is in class scope
  {
        vector<int> v;
        int r = 0;                           // r is local
        for (int i = 0; i<v.size(); ++i)     // i is in statement scope
             r = max(r,abs(v[i]));
        // no i here
        return r;
  }
  // no r here
private:
  vector<int> v;                             // v is in class scope
};
// no v here
```


Scope

```
// get max and abs from algorithm and cstlib
// no r, i, or v here
class My_vector {
public:
  int largest_buggy()                        // largest is in class scope
  {
        vector<int> v;                       // redeclare v, content is lost
        int r = 0;                           // r is local
        for (int i = 0; i<v.size(); ++i)     // i is in statement scope
             r = max(r,abs(v[i]));
        // no i here
        return r;
  }
  // no r here
private:
  vector<int> v;                             // v is in class scope
};
// no v here
```

if we do like this the returned r will be 0. That's because the local v is such that: v.size() = 0
(since it's not init.)

References

- Lippman Chapters 1 & 7